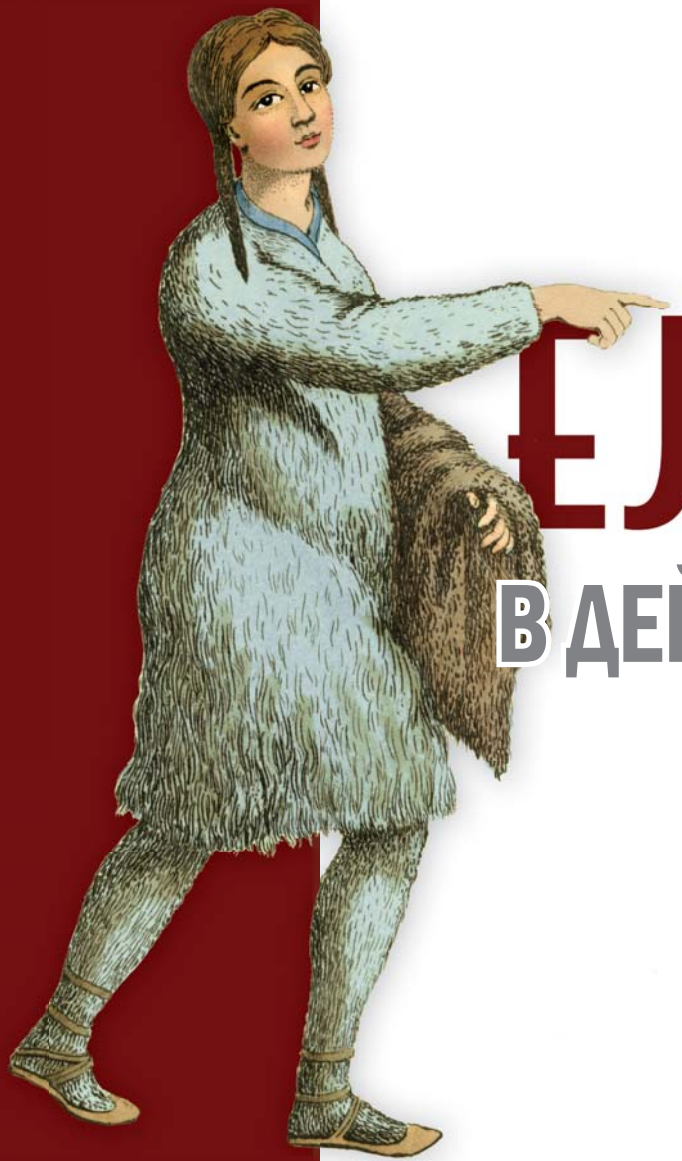
Дебу Панда
Реза Рахман
Райан Купрак
Майкл Ремижан



EJB 3
В ДЕЙСТВИИ

ДМК ПРЕСС
издательство

MANNING

УДК 004.455.2
ББК 32.973.41
П16

П16 Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан
EJB 3 в действии. / Пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 618 с.: ил.

ISBN 978-5-97060-135-8

Фреймворк EJB 3 предоставляет стандартныий способ оформления прикладной логики в виде управляемых модулей, которые выполняются на стороне сервера, упрощая тем самым создание, сопровождение и расширение приложений Java EE. Версия EJB 3.2 включает большее число расширений и более тесно интегрируется с другими технологиями Java, такими как CDI, делая разработку еще проще. Книга знакомит читателя с EJB на многочисленных примерах кода, сценариях из реальной жизни и иллюстрациях. Помимо основ в ней описываются некоторые особенности внутренней реализации, наиболее эффективные приемы использования, шаблоны проектирования, даются советы по оптимизации производительности и различные способы доступа, включая веб-службы, службы REST и веб-сокеты.

Издание предназначено программистам, уже знающим язык Java. Опыт работы с EJB или Java EE не требуется.

УДК 004.455.2
ББК 32.973.41

ISBN 978-1-93518-299-3 (англ.)
ISBN 978-5-97060-135-8 (рус.)

ОГЛАВЛЕНИЕ

Предисловие **14**
Благодарности 15
О книге 18
Структура книги 19
Загружаемый исходный код 20
Соглашения по оформлению исходного кода 20
Автор в сети 20
О названии 21
Об авторах 21
Об илллюстрации на обложке 22

ЧАСТЬ I
Обзор ландшафта EJB **23**

Глава 1. Что такое EJB 3 ..... **24**
1.1. Обзор EJB 25
1.1.1. EJB как модель компонентов 26
1.1.2. Службы компонентов EJB 26
1.1.3. Многоуровневые архитектуры и EJB 28
1.1.4. Почему стоит выбрать EJB 3? 32
1.2. Основы типов EJB 34
1.2.1. Сеансовые компоненты 34
1.2.2. Компоненты, управляемые сообщениями 35
1.3. Связанные спецификации 35
1.3.1. Сущности и Java Persistence API 35
1.3.2. Контексты и внедрение зависимостей для Java EE 37
1.4. Реализации EJB 37
1.4.1. Серверы приложений 38
1.4.2. EJB Lite 39
1.4.3. Встраиваемые контейнеры 40
1.4.4. Использование EJB 3 в Tomcat 40
1.5. Превосходные инновации 41
1.5.1. Пример «Hello User» 41
1.5.2. Аннотации и XML 42
1.5.3. Значения по умолчанию и явные настройки 43
1.5.4. Внедрение зависимостей и поиск в JNDI 44
1.5.5. CDI и механизм внедрения в EJB 45
1.5.6. Тестируемость компонентов POJO 45

1.6. Новшества в EJB 3.2 46
1.6.1. Поддержка EJB 2 теперь является необязательной 46
1.6.2. Усовершенствования в компонентах, управляемых сообщениями 46
1.6.3. Усовершенствования в сеансовых компонентах с сохранением состояния 47
1.6.4. Упрощение локальных интерфейсов компонентов без сохранения состояния 48
1.6.5. Усовершенствования в TimerService API 49
1.6.6. Усовершенствования в EJBContainer API 49
1.6.7. Группы EJB API 49
1.7. В заключение 50

Глава 2. Первая проба EJB 51
2.1. Введение в приложение ActionBazaar 52
2.1.1. Архитектура 52
2.1.2. Решение на основе EJB 3 54
2.2. Реализация прикладной логики с применением EJB 3 55
2.2.1. Использование сеансовых компонентов без сохранения состояния 56
2.2.2 Использование сеансовых компонентов с сохранением состояния 58
2.2.3. Модульное тестирование компонентов EJB 3 63
2.3. Использование CDI с компонентами EJB 3 64
2.3.1. Использование CDI с JSF 2 и EJB 3 65
2.3.2. Использование CDI с EJB 3 и JPA 2 68
2.4. Использование JPA 2 с EJB 3 70
2.4.1. Отображение сущностей JPA 2 в базу данных 71
2.4.2. Использование EntityManager 72
2.5. В заключение 74

ЧАСТЬ II
Компоненты EJB 75

Глава 3. Реализация прикладной логики с помощью сеансовых компонентов 76
3.1. Знакомство с сеансовыми компонентами 77
3.1.1. Когда следует использовать сеансовые компоненты 78
3.1.2. Состояние компонента и типы сеансовых компонентов 80
3.2. Сеансовые компоненты без сохранения состояния 83
3.2.1. Когда следует использовать сеансовые компоненты без сохранения состояния 83
3.2.2. Организация компонентов в пулы 84
3.2.3. Пример BidService 86
3.2.4. Применение аннотации @Stateless 89
3.2.5. Прикладные интерфейсы компонентов 90
3.2.6. События жизненного цикла 93
3.2.7. Эффективное использование сеансовых компонентов без сохранения состояния 96
3.3. Сеансовые компоненты с сохранением состояния 97
3.3.1. Когда следует использовать сеансовые компоненты с сохранением состояния 98

3.3.2. Пассивация компонентов ... 99
3.3.3. Сеансовые компоненты с сохранением состояния в кластере ... 100
3.3.4. Пример реализации создания учетной записи ... 100
3.3.5. Применение аннотации @Stateful ... 104
3.3.6. Прикладные интерфейсы компонентов ... 105
3.3.7. События жизненного цикла ... 105
3.3.8. Эффективное использование сеансовых компонентов с сохранением состояния ... 107
3.4. Сеансовые компоненты-одиночки ... 109
3.4.1. Когда следует использовать сеансовые компоненты-одиночки ... 110
3.4.2. Пример реализации «товара дня» в ActionBazaar ... 111
3.4.3. Применение аннотации @Singleton ... 113
3.4.4. Управление конкуренцией в компоненте-одиночке ... 114
3.4.5. Прикладной интерфейс компонента ... 117
3.4.6. События жизненного цикла ... 118
3.4.7. Аннотация @Startup ... 119
3.4.8. Эффективное использование сеансовых компонентов-одиночек ... 120
3.5. Асинхронные сеансовые компоненты ... 122
3.5.1. Основы асинхронного вызова ... 122
3.5.2. Когда следует использовать асинхронные сеансовые компоненты ... 123
3.5.3. Пример компонента ProcessOrder ... 124
3.5.4. Применение аннотации @Asynchronous ... 126
3.5.5. Применение интерфейса Future ... 127
3.5.6. Эффективное использование асинхронных сеансовых компонентов ... 127
3.6. В заключение ... 128

Глава 4. Обмен сообщениями и разработка компонентов MDB ... 130

4.1. Концепции обмена сообщениями ... 131
4.1.1. Промежуточное ПО передачи сообщений ... 131
4.1.2. Обмен сообщениями в ActionBazaar ... 132
4.1.3. Модели обмена сообщениями ... 134
4.2. Введение в JMS ... 136
4.2.1. Интерфейс JMS Message ... 138
4.3. Использование компонентов MDB ... 140
4.3.1. Когда следует использовать обмен сообщениями и компоненты MDB ... 141
4.3.2. Почему следует использовать MDB? ... 141
4.3.3. Разработка потребителя сообщений с применением MDB ... 143
4.3.4. Применение аннотации @MessageDriven ... 144
4.3.5. Реализация интерфейса MessageListener ... 145
4.3.6. Использование параметра ActivationConfigProperty ... 146
4.3.7. События жизненного цикла ... 149
4.3.8. Отправка сообщений JMS из компонентов MDB ... 151
4.3.9. Управление транзакциями MDB ... 152
4.4. Приемы использования компонентов MDB ... 153
4.5. В заключение ... 155

Глава 5. Контекст EJB времени выполнения, внедрение зависимостей и сквозная логика 157
5.1. Контекст EJB 157
5.1.1. Основы контекста EJB 158
5.1.2. Интерфейсы контекста EJB 159
5.1.3. Доступ к контейнеру через контекст EJB 160
5.2. Использование EJB DI и JNDI 161
5.2.1. Пример использования JNDI в EJB 162
5.2.2. Как присваиваются имена компонентам EJB 166
5.2.3. Внедрение зависимостей с применением @EJB 169
5.2.4. Когда следует использовать внедрение зависимостей EJB 170
5.2.5. Аннотация @EJB в действии 171
5.2.6. Внедрение ресурсов с помощью аннотации @Resource 173
5.2.7. Когда следует использовать внедрение ресурсов 175
5.2.8. Аннотация @Resource в действии 175
5.2.9. Поиск ресурсов и компонентов EJB в JNDI 178
5.2.10. Когда следует использовать поиск в JNDI 180
5.2.11. Контейнеры клиентских приложений 180
5.2.12. Встраиваемые контейнеры 181
5.2.13. Эффективный поиск и внедрение компонентов EJB 183
5.2.14. Механизмы внедрения EJB и CDI 184
5.3. AOP в мире EJB: интерцепторы 185
5.3.1. Что такое AOP? 185
5.3.2. Основы интерцепторов 186
5.3.3. Когда следует использовать интерцепторы 187
5.3.4. Порядок реализации интерцепторов 187
5.3.5. Определение интерцепторов 188
5.3.6. Интерцепторы в действии 192
5.3.7. Эффективное использование интерцепторов 198
5.3.8. Интерцепторы CDI и EJB 199
5.4. В заключение 205

Глава 6. Транзакции и безопасность 206
6.1. Знакомство с транзакциями 207
6.1.1. Основы транзакций 208
6.1.2. Транзакции в Java 210
6.1.3. Транзакции в EJB 212
6.1.4. Когда следует использовать транзакции 214
6.1.5. Как реализованы транзакции EJB 215
6.1.6. Двухфазное подтверждение 217
6.1.7. Производительность JTA 218
6.2. Транзакции, управляемые контейнером 219
6.2.1. Досрочное оформление заказов с применением модели CMT 219
6.2.2. Аннотация @TransactionManagement 220
6.2.3. Аннотация @TransactionAttribute 221
6.2.4. Откат транзакций в модели CMT 224
6.2.5. Транзакции и обработка исключений 226
6.2.6. Синхронизация с сеансом 228
6.2.7. Эффективное использование модели CMT 228

6.3. Транзакции, управляемые компонентами ... 229
6.3.1. Досрочное оформление заказов с применением модели BMT ... 230
6.3.2. Получение экземпляра UserTransaction ... 231
6.3.3. Использование интерфейса UserTransaction ... 232
6.3.4. Эффективное использование модели BMT ... 234
6.4. Безопасность EJB ... 234
6.4.1. Аутентификация и авторизация ... 235
6.4.2. Пользователи, группы и роли ... 236
6.4.3. Как реализована поддержка безопасности в EJB ... 237
6.4.4. Декларативное управление безопасностью в EJB ... 241
6.4.5. Программное управление безопасностью в EJB ... 243
6.4.6. Эффективное использование поддержки безопасности в EJB ... 246
6.5. В заключение ... 247

Глава 7. Планирование и таймеры ... 249
7.1. Основы планирования ... 250
7.1.1. Возможности Timer Service ... 250
7.1.2. Таймауты ... 253
7.1.3. Cron ... 253
7.1.4. Интерфейс Timer ... 254
7.1.5. Типы таймеров ... 256
7.2. Декларативные таймеры ... 257
7.2.1. Аннотация @Schedule ... 257
7.2.2. Аннотация @Schedules ... 258
7.2.3. Параметры аннотации @Schedule ... 258
7.2.4. Пример использования декларативных таймеров ... 259
7.2.5. Синтаксис правил в стиле cron ... 260
7.3. Программные таймеры ... 263
7.3.1. Знакомство с программными таймерами ... 263
7.3.2. Пример использования программных таймеров ... 265
7.3.3. Эффективное использование программных таймеров EJB ... 267
7.4. В заключение ... 268

Глава 8. Компоненты EJB как веб-службы ... 270
8.1. Что такое «веб-служба»? ... 271
8.1.1. Свойства веб-служб ... 271
8.1.2. Транспорты ... 272
8.1.3. Типы веб-служб ... 272
8.1.4. Java EE API для веб-служб ... 273
8.1.5. Веб-службы и JSF ... 274
8.2. Экспортирование компонентов EJB с использованием SOAP (JAX-WS) ... 274
8.2.1. Основы SOAP ... 274
8.2.2. Когда следует использовать службы SOAP ... 279
8.2.3. Когда следует экспортировать компоненты EJB в виде веб-служб SOAP ... 280
8.2.4. Веб-служба SOAP для ActionBazaar ... 281
8.2.5. Аннотации JAX-WS ... 286
8.2.6. Эффективное использование веб-служб SOAP в EJB ... 290
8.3. Экспортирование компонентов EJB с использованием REST (JAX-RS) ... 292

8.3.1. Основы REST ... 293
8.3.2. Когда следует использовать REST/JAX-RS ... 296
8.3.3. Когда следует экспортировать компоненты EJB в виде веб-служб REST ... 297
8.3.4. Веб-служба REST для ActionBazaar ... 298
8.3.5. Аннотации JAX-RS ... 302
8.3.6. Эффективное использование веб-служб REST в EJB ... 307
8.4. Выбор между SOAP и REST ... 308
8.5. В заключение ... 310

ЧАСТЬ III
Использование EJB совместно с JPA и CDI ... 311

Глава 9. Сущности JPA ... 312
9.1. Введение в JPA ... 313
9.1.1. Несоответствие интерфейсов ... 313
9.1.2. Взаимосвязь между EJB 3 и JPA ... 314
9.2. Предметное моделирование ... 315
9.2.1. Введение в предметное моделирование ... 315
9.2.2. Предметная модель приложения ActionBazaar ... 315
9.3. Реализация объектов предметной области с помощью JPA ... 320
9.3.1. Аннотация @Entity ... 320
9.3.2. Определение таблиц ... 322
9.3.3. Отображение свойств в столбцы ... 325
9.3.4. Типы представления времени ... 330
9.3.5. Перечисления ... 331
9.3.6. Коллекции ... 332
9.3.7. Определение идентичности сущностей ... 334
9.3.8. Генерирование значений первичных ключей ... 339
9.4. Отношения между сущностями ... 343
9.4.1. Отношение «один к одному» ... 344
9.4.2. Отношения «один ко многим» и «многие к одному» ... 346
9.4.3. Отношение «многие ко многим» ... 349
9.5. Отображение наследования ... 350
9.5.1. Стратегия единой таблицы ... 351
9.5.2. Стратегия соединения таблиц ... 353
9.5.3. Стратегия отдельных таблиц для каждого класса ... 354
9.6. В заключение ... 357

Глава 10. Управление сущностями ... 358
10.1. Введение в использование EntityManager ... 358
10.1.1. Интерфейс EntityManager ... 359
10.1.2. Жизненный цикл сущностей ... 361
10.1.3. Контекст сохранения, области видимости и EntityManager ... 364
10.1.4. Использование EntityManager в ActionBazaar ... 366
10.1.5. Внедрение EntityManager ... 367
10.1.6. Внедрение EntityManagerFactory ... 369
10.2. Операции с хранилищем ... 371
10.2.1. Сохранение сущностей ... 372

10.2.2. Извлечение сущностей по ключу ... 373
10.2.3. Изменение сущностей ... 379
10.2.4. Удаление сущностей ... 382
10.3. Запросы сущностей ... 384
10.3.1. Динамические запросы ... 385
10.3.2. Именованные запросы ... 385
10.4. В заключение ... 386

Глава 11. JPQL ... 387
11.1. Введение в JPQL ... 387
11.1.1. Типы инструкций ... 388
11.1.2. Предложение FROM ... 390
11.1.3. Инструкция SELECT ... 401
11.1.4. Управление результатами ... 404
11.1.5. Соединение сущностей ... 405
11.1.6. Операции массового удаления и изменения ... 408
11.2. Запросы Criteria ... 409
11.2.1. Метамодели ... 410
11.2.2. CriteriaBuilder ... 413
11.2.3. CriteriaQuery ... 414
11.2.4. Корень запроса ... 415
11.2.5. Предложение FROM ... 419
11.2.6. Предложение SELECT ... 419
11.3. Низкоуровневые запросы ... 422
11.3.1. Динамические SQL-запросы ... 423
11.3.2. Именованные SQL-запросы ... 424
11.3.3. Хранимые процедуры ... 425
11.4. В заключение ... 429

Глава 12. Использование CDI в EJB 3 ... 430
12.1. Введение в CDI ... 431
12.1.1. Службы CDI ... 433
12.1.2. Отношения между CDI и EJB 3 ... 436
12.1.3. Отношения между CDI и JSF 2 ... 437
12.2. Компоненты CDI ... 437
12.2.1. Как пользоваться компонентами CDI ... 438
12.2.2. Именование компонентов и их разрешение в выражениях EL ... 439
12.2.3. Области видимости компонентов ... 440
12.3. Следующее поколение механизмов внедрения зависимостей ... 443
12.3.1. Внедрение с помощью @Inject ... 443
12.3.2. Фабричные методы ... 445
12.3.3. Квалификаторы ... 448
12.3.4. Методы уничтожения ... 449
12.3.5. Определение альтернатив ... 450
12.4. Интерцепторы и декораторы ... 453
12.4.1. Привязка интерцепторов ... 453
12.4.2. Декораторы ... 456
12.5. Стереотипы ... 457
12.6. Внедрение событий ... 459

12.7. Диалоги ... 461
12.8. Эффективное использование CDI в EJB 3 ... 467
12.9. В заключение ... 469

ЧАСТЬ IV
Ввод EJB в действие ... 471

Глава 13. Упаковка приложений EJB 3 ... 472
13.1. Упаковка приложений ... 472
13.1.1. Строение системы модулей Java EE ... 475
13.1.2. Загрузка модулей Java EE ... 476
13.2. Загрузка классов ... 478
13.2.1. Основы загрузки классов ... 478
13.2.2. Загрузка классов в приложениях Java EE ... 478
13.2.3. Зависимости между модулями Java EE ... 481
13.3. Упаковка сеансовых компонентов и компонентов, управляемых сообщениями ... 483
13.3.1. Упаковка EJB-JAR ... 483
13.3.2. Упаковка компонентов EJB в модуль WAR ... 485
13.3.3. XML против аннотаций ... 488
13.3.4. Переопределение настроек, указанных в аннотациях ... 492
13.3.5. Определение интерцепторов по умолчанию ... 493
13.4. Упаковка сущностей JPA ... 494
13.4.1. Модуль доступа к хранимым данным ... 494
13.4.2. Описание модуля доступа к хранимым данным в persistence.xml ... 496
13.5. Упаковка компонентов CDI ... 498
13.5.1. Модули CDI ... 498
13.5.2. Дескриптор развертывания beans.xml ... 499
13.5.3. Атрибут bean-discovery-mode ... 500
13.6. Эффективные приемы и типичные проблемы развертывания ... 501
13.6.1. Эффективные приемы упаковки и развертывания ... 501
13.6.2. Решение типичных проблем развертывания ... 503
13.7. В заключение ... 504

Глава 14. Использование веб-сокетов с EJB 3 ... 505
14.1. Ограничения схемы взаимодействий «запрос/ответ» ... 505
14.2. Введение в веб-сокеты ... 507
14.2.1. Основы веб-сокетов ... 507
14.2.2. Веб-сокеты и AJAX ... 511
14.2.3. Веб-сокеты и Comet ... 513
14.3. Веб-сокеты и Java EE ... 515
14.3.1. Конечные точки веб-сокетов ... 516
14.3.2. Интерфейс Session ... 517
14.3.3. Кодеры и декодеры ... 520
14.4. Веб-сокеты в приложении ActionBazaar ... 523
14.4.1. Использование программных конечных точек ... 526
14.4.2. Использование аннотированных конечных точек ... 530
14.5. Эффективное использование веб-сокетов ... 537
14.6. В заключение ... 539

Глава 15. Тестирование компонентов EJB 541
15.1. Введение в тестирование 541
15.1.1. Стратегии тестирования 542
15.2. Модульное тестирование компонентов EJB 544
15.3. Интеграционное тестирование с использованием EJBContainer 548
15.3.1. Настройка проекта 549
15.3.2. Интеграционный тест 552
15.4. Интеграционное тестирование с применением Arquillian 555
15.4.1. Настройка проекта 556
15.4.2. Интеграционный тест 560
15.5. Приемы эффективного тестирования 563
15.6. В заключение 565

Приложение A. Дескриптор развертывания, справочник 566
A.1. ejb-jar.xml 566
A.1.1. <module-name> 567
A.1.2. <enterprise-beans> 567
A.1.3. Интерцепторы 571
A.1.4. <assembly-descriptor> 571

Приложение B. Введение в Java EE 7 SDK 576
B.1. Установка Java EE 7 SDK 576
B.2. GlassFish Administration Console 581
B.3. Запуск и остановка GlassFish 584
B.4. Запуск приложения «Hello World» 586

Приложение C. Сертификационные экзамены разработчика для EJB 3 590
C.1. Начало процесса сертификации 591
C.2. Порядок прохождения сертификационных испытаний для разработчиков EJB 3 593
C.3. Знания, необходимые для прохождения испытаний 595
C.4. Подготовка к испытаниям 597
C.5. Сертификационные испытания 598

Предметный указатель 600

ГЛАВА 2.
Первая проба EJB

Эта глава охватывает следующие темы:

- приложение ActionBazaar;
- сеансовые компоненты с сохранением и без сохранения состояния в приложении ActionBazaar;
- интеграция CDI и EJB 3;
- сохранение объектов в JPA 2.

В эпоху глобализации изучение технологий с книгой на коленях и клавиатурой под руками, попутно решая практические задачи, стало нормой. Посмотрим правде в глаза – где-то в глубине души вы наверняка предпочли бы пройти подобное «крещение огнем», чем снова и снова тащиться по старым, проторенным дорогам. Эта глава – для храбрых первооткрывателей, живущих внутри нас, желающих заглянуть за горизонт, в новый мир EJB 3.

В первой главе вы осмотрели ландшафт EJB 3 с высоты 6000 метров, из окна сверхзвукового лайнера. Мы дали определение EJB, описали службы и общую структуру EJB 3, а также рассказали, как EJB 3 связана с CDI и JPA 2. В этой главе мы пересядем в маленький разведывательный самолет и пролетим намного ниже. Здесь мы бегло рассмотрим пример решения практической задачи с использованием EJB 3, JPA 2 и CDI. В этом примере будут использоваться компоненты EJB 3 нескольких типов, многоуровневая архитектура и некоторые службы, представленные в главе 1. Вы сами убедитесь, насколько проста и удобна EJB 3 и как быстро можно включить ее в работу.

Если вы не большой любитель обозревать окрестности с высоты, не пугайтесь. Представьте себе эту главу, как первый день на новой работе, когда вы пожимаете руки незнакомцам за соседними столами. В следующих главах вы поближе сойдетесь с новыми коллегами, узнаете об их пристрастиях, предубеждениях и причудах, и научитесь обходить их недостатки. А пока от вас требуется только запомнить их имена.

Опробование примеров кода

С этого момента мы предлагаем не пренебрегать возможностью исследования примеров кода. Вы можете получить полный комплект примеров, загрузив zip-файл со страницы www.manning.com/panda2. Мы также настоятельно рекомендуем установить среду разработки по своему выбору для опробования кода. В этом случае вы сможете следовать за нами, вносить в код свои изменения и запускать его внутри контейнера.

В этой главе мы приступим к решению задачи, которое продолжится в остальных главах, – созданию приложения ActionBazaar. Это воображаемая система, на примере которой мы будем пояснять различные аспекты. В некотором смысле эта книга является учебным примером разработки приложения ActionBazaar с использованием EJB 3. А теперь коротко пройдемся по приложению ActionBazaar, чтобы понять, о чем пойдет речь.

2.1. Введение в приложение ActionBazaar

ActionBazaar – это простой интернет-аукцион, по образу и подобию eBay. Продавцы сдувают пыль с сокровищ, хранящихся у них в подвалах, делают несколько размытых фотографий и посылают их в приложение ActionBazaar. Нетерпеливые покупатели входят в соревновательный раж и перебивают цену друг друга в попытке приобрести сокровища, изображенные на нечетких фотографиях и снабженных описаниями с орфографическими ошибками. Победители платят деньги. Продавцы высылают товар. И все счастливы, ну или почти все.

Как бы нам ни хотелось присвоить себе лавры первооткрывателей, тем не менее, мы должны признать, что впервые идея приложения ActionBazaar была выдвинута в книге «Hibernate in Action» Кристиана Бауэра (Christian Bauer) и Гэвина Кинга (Gavin King) (Manning, 2004), в виде примера приложения CaveatEmptor. Книга «Hibernate in Action» в основном посвящена разработке уровня хранения данных с применением фреймворка Hibernate объектно-реляционного отображения. Позднее эта идея была использована Патриком Лайтбоди (Patrick Lightbody) и Джейсоном Каррера (Jason Carreira) в книге «WebWork in Action» (Manning, 2005) при обсуждении открытого фреймворка уровня представления. Нам показалось, что было бы неплохо использовать эту же идею и в нашей книге, «EJB 3 в действии».

В этом разделе мы представим вам приложение ActionBazaar. Сначала мы определим ограниченную архитектуру ActionBazaar, а затем будем наращивать ее с применением EJB 3. В оставшейся части главы, следующей за этим разделом, мы займемся исследованием некоторых важных функций этих технологий на примерах из приложения ActionBazaar, познакомимся с некоторыми типами компонентов EJB и посмотрим, как они используются в комплексе с CDI и JPA 2.

Итак, начнем со знакомства с требованиями и архитектурой примера.

2.1.1. Архитектура

Для начального знакомства с EJB 3 ограничимся в этой главе небольшим подмножеством функциональности ActionBazaar, начав с предложения цены и закончив

оформлением сделки с победителем. Это подмножество функциональности изображено на рис. 2.1.

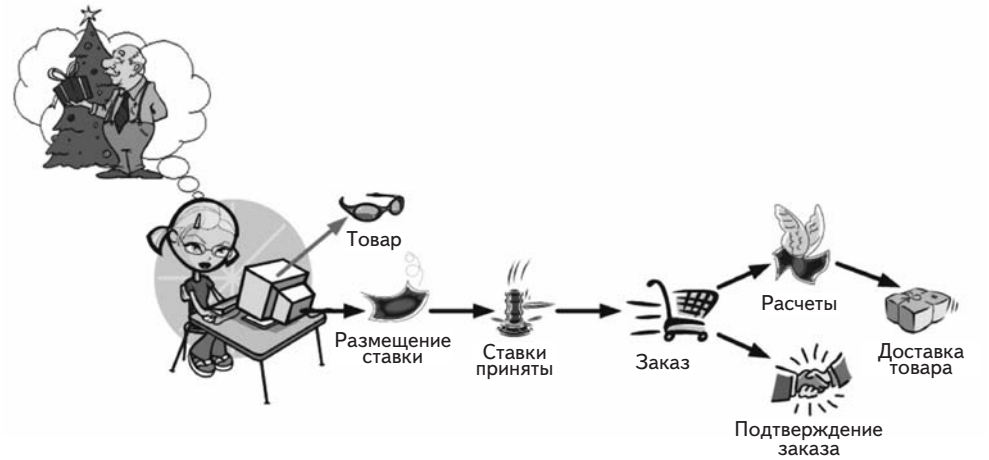


Рис. 2.1. Цепочка операций, выполняемых приложением ActionBazaar, на основе которой мы займемся исследованием EJB 3. Претендент предлагает цену за желаемый товар, выигрывает, заказывает и немедленно получает подтверждение. В процессе подтверждения заказа пользователь вносит информацию о способе оплаты. После подтверждения получения денег продавец организует доставку покупки

Функции, перечисленные на рис. 2.1, представляют основу приложения ActionBazaar. Здесь отсутствуют такие основные функции, как подача объявления о продаже, просмотр списка объявлений и поиск нужного товара. Мы оставили эти функции для обсуждения в дальнейшем. Их реализация включает разработку модели всей предметной области, которая описывается в главе 9, где мы приступим к знакомству с особенностями моделирования предметных областей и механизма хранения данных JPA 2.

Цепочка операций, изображенная на рис. 2.1, начинается с того, что пользователь решил купить некоторый товар и предложить свою цену. Пользователь Дженни (Jenny) выбрала прекрасный подарок для дедушки на Рождество и быстро предлагает цену $5.00. Когда аукцион подходит к концу, побеждает самая высокая ставка. Дженни повезло, и никто не предложил цену выше, чем она, соответственно ее ставка $5.00 выигрывает. Как победителю, Дженни разрешают заказать товар у продавца Джо (Joe). В заказе указывается вся необходимая в таких делах информация – адрес доставки, расчет сумм, необходимых на упаковку и доставку, общая сумма, и так далее. Дженни убеждает свою маму оплатить заказ своей кредитной картой и указывает адрес дедушки, куда следует доставить покупку. Мало чем отличаясь от таких площадок электронных торгов, как Amazon.com и eBay, приложение ActionBazaar не заставляет пользователя ждать, пока пройдут все платежи, перед подтверждением заказа. Вместо этого заказ подтверждается сразу же после приема всех данных и запуска процесса расчетов, который протекает параллель-

но, в фоновом режиме. Дженни получает подтверждение о приеме заказа сразу после щелчка на кнопке **Order** (Заказать). И хотя Дженни этого не видит, процедура списания денег с кредитной карты ее мамы начнется немедленно, в фоновом режиме, сразу после приема подтверждения. Когда процесс расчетов завершится, обеим участникам сделки, Дженни и Джо, будут отправлены электронные письма с уведомлением. После получения уведомления о переводе денег, Джо доставит подарок дедушке Дженни к означенному моменту, как раз перед Рождеством!

В следующем разделе вы увидите, как с помощью EJB 3 можно реализовать компоненты, реализующие эту цепочку операций. Но, прежде чем приступить к изучению схемы решения в следующем разделе, попробуйте наглядно представить, как могли бы располагаться необходимые компоненты в многоуровневой архитектуре EJB. Какое место, по вашему мнению, могли бы занять в этой схеме сеансовые компоненты, CDI, сущности и JPA 2 API, учитывая знания, полученные в главе 1?

2.1.2. Решение на основе EJB 3

На рис. 2.2 показано, как с помощью EJB 3 можно реализовать сценарий работы ActionBazaar, описанный в предыдущем разделе, в традиционной четырехуровневой архитектуре и с применением модели предметной области.

Если внимательно изучить сценарий на рис. 2.2, можно заметить, что пользователь активизирует только две процедуры: добавление ставки и заказ товара в случае победы. Как вы уже наверняка догадались процедуры добавления ставки и заказа реализованы с помощью сеансовых компонентов (`BidService` и `OrderProcessor`) на уровне прикладной логики.

В ходе обеих процедур производится сохранение данных. Компоненту `BidService` нужно добавить в базу данных запись о сделанной ставке. Аналогично компоненту `OrderProcessor` нужно добавить запись с информацией о заказе. Все необходимые изменения в базе данных выполняются с помощью двух сущностей на уровне хранения, действующих под управлением JPA – `Bid` и `Order`. При этом `BidService` использует сущность `Bid`, а `OrderProcessor` – сущность `Order`. Посредством этих сущностей компоненты на уровне прикладной логики используют компоненты `BidDao` и `OrderDao` уровня хранения. Обратите внимание, что объекты DAO доступа к данным не обязательно должны быть компонентами EJB, потому что для решения стоящих перед ними задач не требуются службы, предоставляемые EJB. В действительности, как будет показано далее, объекты DAO являются простейшими Java-объектами, управляемыми механизмом CDI, и не используют никакие службы, кроме внедрения зависимостей. Напомним, что даже при том, что сущности JPA 2 содержат настройки ORM, сами они не участвуют в сохранении непосредственно. Как вы увидите в фактической реализации, для добавления, удаления, изменения и извлечения сущностей объекты DAO используют EntityManager API из JPA 2.

Если ваше мысленное представление достаточно близко совпадает с рис. 2.2, значит и код, который приводится далее, будет вам понятен, даже при том, что вы, возможно, не знакомы с особенностями EJB 3.

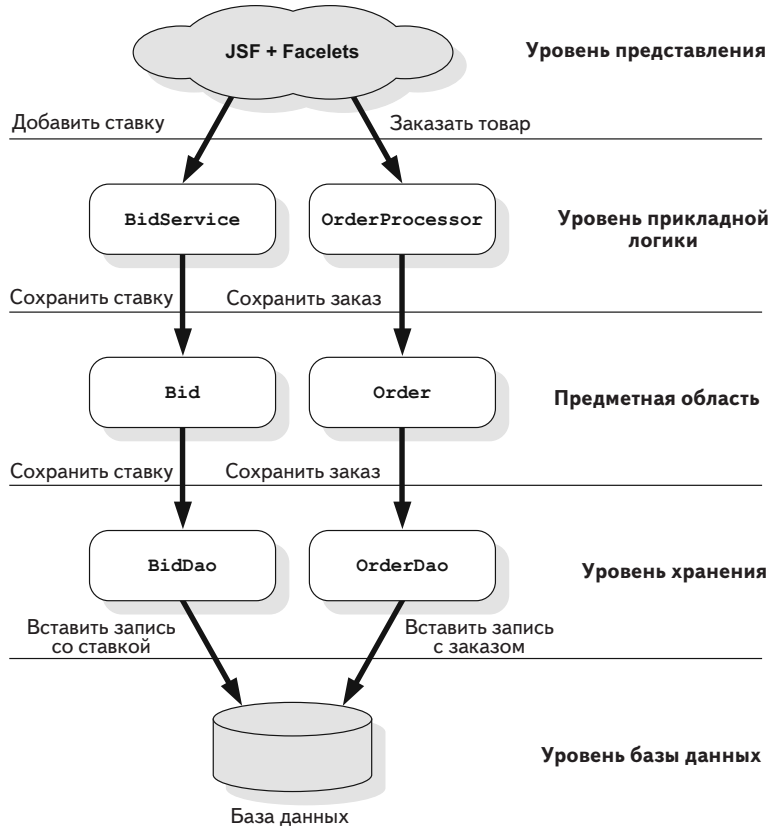


Рис. 2.2. Сценарий работы ActionBazaar реализован с помощью EJB 3. С позиции EJB 3 уровень представления выглядит как аморфное облако (в данном случае реализованное на основе JSF), генерирующее запросы к уровню прикладной логики. Компоненты уровня прикладной логики соответствуют различным операциям в сценарии – размещение ставки и оформление заказа для победившей ставки. Для сохранения информации в базе данных компоненты уровня прикладной логики используют сущности JPA, используя для этого объекты доступа к данным (DAO)

2.2. Реализация прикладной логики с применением EJB 3

Итак, приступим к исследованию решения, начав с уровня прикладной логики, как вы поступили бы в случае действующего приложения. Сеансовые компоненты EJB 3 идеально подходят для реализации процедур моделирования ставки и заказа. По умолчанию они поддерживают транзакции, многопоточную модель выполнения и возможность размещения в пулах – все характеристики, необходимые для постро-

ения прикладного уровня приложения. Сеансовые компоненты – самые простые, и вместе с тем самые универсальные элементы EJB 3, поэтому мы и начнем с них.

Напомним, что существует три разновидности сеансовых компонентов: с сохранением состояния, без сохранения состояния и компоненты-одиночки (singleton). В наших примерах мы будем использовать только сеансовые компоненты с сохранением и без сохранения состояния. Позднее нам также понадобятся компоненты, управляемые сообщениями (Message-Driven Beans, MDB). Для начала исследуем сеансовые компоненты без сохранения состояния, просто потому, что они проще.

2.2.1. Использование сеансовых компонентов без сохранения состояния

Сеансовые компоненты без сохранения состояния часто используются для моделирования операций, которые могут быть выполнены за один вызов метода, таких как добавление ставки в сценарии ActionBazaar. На практике сеансовые компоненты без сохранения состояния используются чаще других на уровне прикладной логики. Метод `addBid`, представленный в листинге 2.1, вызывается веб-уровнем приложения ActionBazaar, когда пользователь решает сделать ставку. Параметр метода, объект `Bid`, представляет саму ставку. Этот объект содержит имя претендента, информацию о товаре, для которого делается ставка, и сумма ставки. Как вы уже знаете, работа этого метода заключается в том, чтобы сохранить объект `Bid` в базе данных. Ближе к концу главы вы увидите, что в действительности объект `Bid` является сущностью JPA 2.

Листинг 2.1. Сеансовый компонент без сохранения состояния BidService

```
@Stateless // Пометить POJO как сеансовый компонент без сохранения состояния
public class DefaultBidService implements BidService {
    @Inject // Внедрить объект DAO
    private BidDao bidDao;
    ...
    public void addBid(Bid bid) {
        bidDao.addBid(bid);
    }
    ...
}

@Local // Пометить интерфейс как локальный
public interface BidService {
    ...
    public void addBid(Bid bid);
    ...
}
```

Первое, на что вы наверняка обратили внимание, насколько простым выглядит код. Класс `DefaultBidService` – это простой Java-объект (Plain Old Java Object, POJO) а интерфейс `BidService` – простой Java-интерфейс (Plain Old Java Interface, POJI). Интерфейс не содержит ничего такого, что вызывало бы сложности при его

реализации в наследующем классе. Единственно примечательной особенностью в листинге 2.1 являются три аннотации – `@Stateless`, `@Local` и `@Inject`.

- `@Stateless` – Аннотация `@Stateless` сообщает контейнеру EJB, что `DefaultBidService` является сеансовым компонентом без сохранения состояния. Встретив такую аннотацию, контейнер автоматически добавит в компонент поддержку многопоточной модели выполнения, транзакций и возможность размещения в пулах. Поддержка многопоточности и транзакций гарантируют возможность использования любых ресурсов, таких как база данных или очередь сообщений, без необходимости вручную писать код для обслуживания конкуренции или транзакций. Поддержка размещения в пулах гарантирует надежность даже при очень высоких нагрузках. При необходимости в компонент можно также добавить поддержку дополнительных служб EJB, таких как безопасность, планирование и интерцепторы.
- `@Local` – Аннотация `@Local`, отмечающая интерфейс `BidService` сообщает контейнеру, что реализация `BidService` может быть доступна локально, посредством интерфейса. Так как компоненты EJB и компоненты, использующие их, обычно находятся в одном и том же приложении, в этом есть определенный смысл. При желании аннотацию `@Local` можно опустить и интерфейс все равно будет интерпретироваться контейнером как локальный. Как вариант, интерфейс можно пометить аннотацией `@Remote` или `@WebService`. В случае применения аннотации `@Remote`, к компоненту будет предоставлен удаленный доступ через механизм Java Remote Method Invocation (RMI), идеально подходящий для организации удаленного доступа со стороны Java-клиентов. Если потребуется организовать удаленный доступ к компонентам EJB из клиентов, реализованных на других платформах, таких как Microsoft .NET или PHP, можно включить поддержку SOAP, применив к интерфейсу или классу компонента аннотацию `@WebService`. Отметьте также, что компоненты EJB вообще могут обходиться без каких-либо интерфейсов.
- `@Inject` – Как вы уже знаете, служба добавления ставки зависит от объекта DAO, осуществляющего сохранение данных. Аннотация `@Inject`, реализуемая механизмом CDI, внедряет объект DAO (не имеющий ничего общего с EJB) в переменную экземпляра `BidService`. Если вы пока не знакомы с механизмом внедрения зависимостей, можете представить себе аннотацию `@Inject`, как инструкцию, выполняющую нечто немного необычное, а если честно, то нечто магическое. Кто-то из вас может задаться вопросом: «А можно ли использовать приватную переменную `bidDao`, которая нигде не инициализируется?»! Если бы контейнер не позаботился об этой переменной, вы получили бы позорное исключение `java.lang.NullPointerException` при попытке вызвать метод `addBid` из листинга 2.1, потому что переменная bidDao содержала бы пустое значение. Интересно отметить, что внедрение зависимости можно представить себе как «нестандартный» способ инициализации переменных. Аннотация `@Inject` заставляет контейнер «иници-

ализировать» переменную `bidDao` соответствующей реализацией DAO до того, как произойдет первое обращение к ней.

Подробнее об отсутствии сохраняемого состояния

Так как результаты вызова метода `addBid` (новая ставка) сохраняются в базе данных, клиенту нет нужды беспокоиться о внутреннем состоянии компонента. Здесь нет никакой необходимости в сохранении состояния компонента, чтобы гарантировать сохранность значений его переменных между вызовами. Именно это свойство и называют «без сохранения состояния» при программировании серверных приложений.

Сеансовый компонент `BidService` может позволить себе роскошь не сохранять свое состояние, потому что операция добавления новой ставки выполняется в один шаг. Однако не все прикладные операции так просты. Когда операция выполняется в несколько этапов, может возникнуть необходимость в сохранении внутреннего состояния, чтобы не потерять связь между этапами, – это типичный прием реализации сложных операций. Сохранение состояния может пригодиться, например, когда пользователь при выполнении некоторого этапа определяет, какой этап будет следующим. Представьте себе мастера настройки, выполняющего свою работу в несколько этапов, опираясь на ответы пользователя. Ввод пользователя сохраняется на каждом этапе работы такого мастера и используется, чтобы определить, какой вопрос задать пользователю на следующем этапе. Сеансовые компоненты с сохранением состояния стараются сделать сохранение состояния серверного приложения максимально простым.

Это все, что мы хотели сказать о сеансовых компонентах без сохранения состояния и о службе добавления новой ставки. Теперь давайте обратим наше внимание на процедуру обработки заказа, где как раз требуется сохранять состояние. Немного ниже мы также посмотрим, как служба создания новой ставки используется механизмом JSF, и как выглядит реализация объектов DAO.

2.2.2 Использование сеансовых компонентов с сохранением состояния

В отличие от сеансовых компонентов без сохранения состояния, компоненты с сохранением состояния гарантируют восстановление внутреннего состояния компонента при следующем обращении клиента. Контейнер обеспечивает такую возможность, управляя сеансами и реализуя сохранение.

Управление сеансом

Во-первых, контейнер гарантирует клиенту возможность повторно обращаться к выделенному для него компоненту и вызывать разные его методы. Представьте себе такой компонент, как коммутационный телефонный узел, который будет вас соединять с одним и тем же представителем службы поддержки в пределах некоторого периода времени (такой период называется сеансом).

Во-вторых, контейнер гарантирует сохранность переменных экземпляра на протяжении всего сеанса, не требуя от разработчика писать какой-то дополнительный код для этого. Если продолжить аналогию со службой поддержки, контейнер гарантирует, что информация о вас и вся история ваших звонков в заданный период времени автоматически будет появляться на экране перед представителем службы поддержки при каждом вашем звонке. Операция оформления заказа в приложении ActionBazaar является наглядным примером применения сеансового компонента с сохранением состояния, потому что она выполняется в четыре этапа, каждый из которых грубо соответствует отдельному экрану, которые видит пользователь:

1. Выбор товара для заказа – Процедура оформления заказа начинается с щелчка пользователя на кнопке **Order Item** (Оформить заказ) на странице со списком выигравших ставок, после чего товар автоматически добавляется в заказ.
2. Определение информации о доставке, включая способ доставки, адрес, страховка и так далее. Информация о пользователе может храниться в истории предыдущих заказов, в том числе и некоторые умолчания. Пользователь может просмотреть историю и использовать информацию из предыдущих заказов. При этом на экране должна отображаться только информация, соответствующая выбранному товару (например, продавец может быть готов отправить товар только ограниченным числом способов и по ограниченному кругу адресов). После ввода информации о доставке автоматически рассчитывается ее стоимость.
3. Определение информации о порядке оплаты, такой как номер кредитной карты и адрес оплаты. Для данных о порядке оплаты также поддерживаются функции истории/умолчаний/фильтрации, как и для информации о доставке.
4. Подтверждение заказа после просмотра окончательной формы заказа, включающей общую стоимость.

Все эти этапы изображены на рис. 2.3. При использовании компонента с сохранением состояния, данные, вводимые пользователем на каждом этапе (а также внутренние данные, видеть которые пользователю совсем необязательно), могут накапливаться в переменных компонента до завершения процедуры.

Теперь, когда вы узнали все необходимое, давайте посмотрим, как это реализуется на практике.

Реализация решения

Следующий пример демонстрирует возможную реализацию процедуры оформления заказа в ActionBazaar в виде компонента `DefaultOrderProcessor`. Как видно из примера, `DefaultOrderProcessor` примерно отражает модель выполнения процедуры заказа. Методы в листинге приводятся в порядке, соответствующем последовательности их вызова из страниц JSF. Каждый из них реализует отдельный этап процесса оформления заказа. Методы `setBidder` и `setItem` вызываются уровнем представления в самом начале процедуры, когда пользователь щелкает на кнопке **Order** (Заказать). Клиентом, вероятнее всего, является теку-

щий зарегистрированный пользователь, а товаром – текущий выбранный товар. В методе `setBidder` компонент извлекает историю заказов с адресами доставки и информацией о платежах данного клиента, и сохраняет эти данные вместе с информацией о клиенте. В методе `setItem` выполняется фильтрация данных доставке и платежах, и оставляются только те данные, что могут быть применены к текущему товару. Сам товар также сохраняется в переменной экземпляра для использования на следующих этапах.

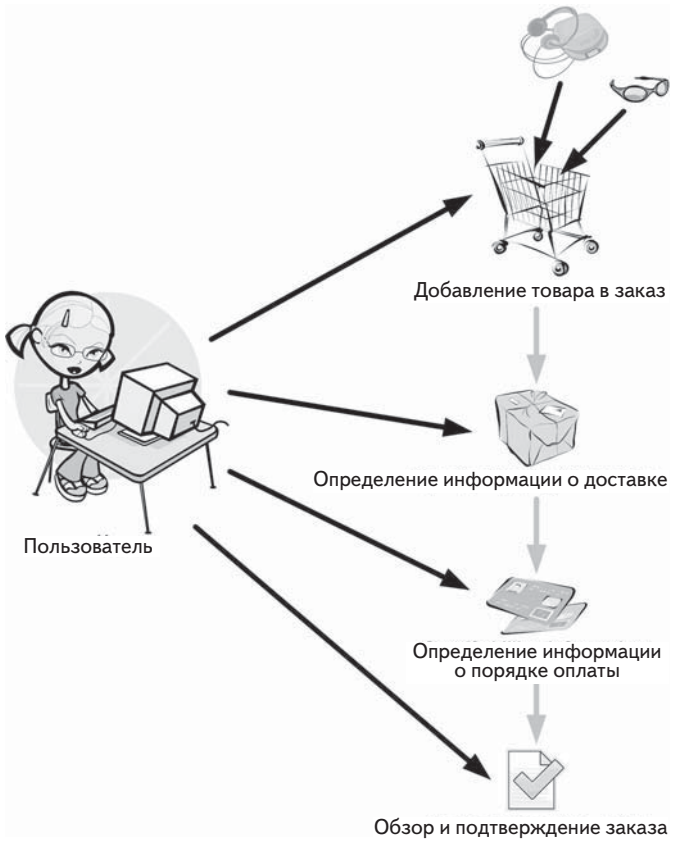


Рис. 2.3. Чтобы сделать процесс управляемым, приложение ActionBazaar разбивает его на несколько этапов. На первом этапе пользователь выбирает товар для заказа, на втором определяет адрес и способ доставки, на третьем указывает порядок оплаты. Завершается этот процесс обзором заключительной формы заказа и подтверждением

Следующее, что делает уровень JSF – запрашивает у пользователя адрес и способ доставки заказа. Для большего удобства, уровень представления предложит пользователю использовать любые допустимые сведения, указанные им в прошлом. Сохраненная информация о доставке извлекается с помощью метода `getShippingChoices`. Когда пользователь выберет элемент из истории или введет новые сведения, данные передаются обратно компоненту оформления заказа, через вызов метода `setShipping`. После установки информации о доставке выполняется сохранение истории клиента, если это необходимо. Компонент оформления заказа также немедленно рассчитает стоимость доставки и добавит ее во

внутреннюю переменную. Уровень JSF может получить эти сведения вызовом метода `getShipping`. Аналогично обрабатывается информация о порядке оплаты – с использованием методов `getBillingChoices` и `setBilling`.

В самом конце вызывается метод `placeOrder`, уже после того, как пользователь проверил заказ и щелкнул на кнопке **Confirm Order** (Подтвердить заказ). Метод `placeOrder` создает и заполняет фактический объект `Order` и пытается получить оплату с клиента, включая стоимость товара, его доставки, страховки и другие выплаты. Сделать это можно множеством способов, например, выполнить перевод средств с кредитной карты или со счета в банке. После попытки перевода денег компонент извещает покупателя и продавца о результатах. Если перевод состоялся, продавцу остается отправить товар по указанному адресу, указанным способом. Если попытка оплаты потерпела неудачу, клиент должен исправить, возможно, ошибочную информацию о способе оплаты и отправить ее вновь. Наконец, компонент сохраняет запись в базе данных, отражающую результат попытки оплаты, как показано в листинге 2.2.

Листинг 2.2. Сеансовый компонент с сохранением состояния OrderProcessor

```
@Stateful // ❶ Пометить POJO как компонент с сохранением состояния
public class DefaultOrderProcessor implements OrderProcessor {
    ...
    private Bidder bidder;                    // ❷ Определения
    private Item item;                        //    переменных
    private Shipping shipping;                //    экземпляра
    private List<Shipping> shippingChoices; //    с состоянием
    private Billing billing;                  //
    private List<Billing> billingChoices;     //

    public void setBidder(Bidder bidder) {
        this.bidder = bidder;
        this.shippingChoices = getShippingHistory(bidder);
        this.billingChoices = getBillingHistory(bidder);
    }

    public void setItem(Item item) {
        this.item = item;
        this.shippingChoices = filterShippingChoices(shippingChoices, item);
        this.billingChoices = filterBillingChoices(billingChoices, item);
    }

    public List<Shipping> getShippingChoices() {
        return shippingChoices;
    }

    public void setShipping(Shipping shipping) {
        this.shipping = shipping;
        updateShippingHistory(bidder, shipping);
        shipping.setCost(calculateShippingCost(shipping, item));
    }

    public Shipping getShipping() {
        return shipping;
```

```
    }

    public List<Billing> getBillingChoices() {
        return billingChoices;
    }

    public void setBilling(Billing billing) {
        this.billing = billing;
        updateBillingHistory(bidder, billing);
    }

    @Asynchronous // ❸ Объявляет метод асинхронным
    @Remove       // ❹ Объявляет метод заключительным
    public void placeOrder() {
        Order order = new Order();
        order.setBidder(bidder);
        order.setItem(item);
        order.setShipping(shipping);
        order.setBilling(billing);
        try {
            bill(order);
            notifyBillingSuccess(order);
            order.setStatus(OrderStatus.COMPLETE);
        } catch (BillingException be) {
            notifyBillingFailure(be, order);
            order.setStatus(OrderStatus.BILLING_FAILED);
        } finally {
            saveOrder(order);
        }
    }
    ...
}
```

Как видите, разработка компонентов с сохранением и без сохранения состояния отличается совсем немного. С точки зрения разработчика единственное отличие состоит в том, что класс `DefaultOrderProcessor` помечен аннотацией `@Stateful` вместо `@Stateless` ❶. Однако, как вы уже знаете, за кулисами различия выглядят гораздо более существенными и проявляются в том, как контейнер обслуживает взаимоотношения между клиентом и значениями, хранящимися в переменных экземпляра компонента ❷. Аннотация `@Stateful` также служит своеобразным сигналом для разработчика клиентской части, сообщающим, чего можно ожидать от компонента, если его поведение будет сложно определить из API и документации.

Аннотация `@Asynchronous` ❸ перед методом `placeOrder` обеспечивает возможность асинхронного выполнения метода. Это означает, что компонент будет возвращать управление клиенту немедленно, а метод продолжит выполнение в фоновом легковесном процессе. Это важная особенность в данном случае, потому что процесс оплаты потенциально может занять продолжительное время. Вместо того, чтобы заставлять пользователя ждать завершения процесса оплаты, благодаря аннотации `@Asynchronous` пользователь немедленно получит подтверждение о приеме заказа, а заключительный этап процесса оформления заказа сможет завершиться в фоновом режиме.

Обратите также внимание на аннотацию `@Remove` ❹ перед методом `placeOrder`. Хотя эта аннотация является необязательной, она играет важную роль в обеспечении высокой производительности в серверных приложениях. Аннотация `@Remove` отмечает окончание процедуры, моделируемой компонентом с сохранением состояния. В данном случае код сообщает контейнеру, что после вызова метода `placeOrder` нет необходимости продолжать поддерживать сеанс с клиентом. Если не сообщить контейнеру, что вызов этого метода означает окончание процедуры, контейнер будет ждать достаточно долго, пока не примет решение, что время истекло и сеанс можно безопасно закрыть. А так как гарантируется, что компоненты с сохранением состояния будут доступны клиенту на протяжении всего сеанса, это может означать хранение большого объема фактически ненужных данных в драгоценной памяти сервера в течение довольно длительного периода времени!

2.2.3. Модульное тестирование компонентов EJB 3

Появление возможности использовать EJB 3 в окружении Java SE стало одним из интереснейших событий в EJB 3.1. Как уже говорилось в главе 1, это достигается с помощью контейнеров EJB 3, которые могут встраиваться в любое окружение времени выполнения Java. Хотя такое было возможно и прежде, с помощью нестандартных встраиваемых контейнеров, таких как OpenEJB для Java EE 5, тем не менее, спецификация EJB 3.1 объявила поддержку таких контейнеров обязательной для всех реализаций.

Встроенные контейнеры особенно полезны в модульном тестировании компонентов EJB 3 с применением таких фреймворков, как JUnit и TestNG. Обеспечение надежности модульного тестирования компонентов EJB 3 является основной задачей таких проектов, как Arquillian. В листинге 2.3 показано, как легко протестировать сеансовый компонент `OrderProcessor` с помощью JUnit. Данный модульный тест имитирует последовательность действий клиента с помощью объектов-имитаций пользователя и товара.

Листинг 2.3. Клиент сеансового компонента с сохранением состояния

```
@RunWith(Arquillian.class) // ❶ Интегрировать Arquillian в JUnit
public class OrderProcessorTest {
    @Inject                // ❷ Внедрить экземпляр компонента
    private OrderProcessor orderProcessor;
    ...
    @Test                  // ❸ Выполняемый тест
    public void testOrderProcessor {
        // Имитация клиента
        Bidder bidder = (Bidder) userService.getUser(new Long(100));
        // Имитация товара
        Item item = itemService.getItem(new Long(200));
        orderProcessor.setBidder(bidder);
        orderProcessor.setItem(item);

        // Получить историю доставки для клиента
        List<Shipping> shippingChoices = orderProcessor.getShippingChoices();

        // Выбрать первый элемент списка
```

```
        orderProcessor.setShipping(shippingChoices.get(0));

        // Получить историю платежей для клиента
        List<Billing> billingChoices = orderProcessor.getBillingChoices();

        // Выбрать первый элемент списка
        orderProcessor.setBilling(billingChoices.get(0));

        // Завершить процедуру и сеанс
        orderProcessor.placeOrder();

        // Ждать некоторое время перед переносом обработки
        // заказа в отдельный процесс
    }
}
```

Аннотация `@RunWith` ❶ сообщает фреймворку JUnit, что он должен запустить Arquillian в ходе тестирования. Arquillian управляет жизненным циклом встраиваемого контейнера EJB – контейнер запускается перед началом тестирования и завершается по его окончании. Затем Arquillian развертывает тестируемые компоненты во встроенном контейнере – мы опустили код, осуществляющий развертывание, но вы можете увидеть его в загружаемых примерах. Arquillian также отвечает за внедрение компонента управления процедурой оформления заказа в тест ❷, а также всех его зависимостей.

Сам тест достаточно прост и понятен. Во-первых, он отыскивает объекты-имитации клиента и товара (используя пару других служб EJB). На языке модульного тестирования объекты, представляющие клиента и товар, называются частями испытуемого набора данных. Затем выполняется объекты-имитации делаются частью процесса оформления заказа. Тест также имитирует извлечение истории доставки и оплаты клиента и устанавливает необходимые параметры. в обоих случаях тест выбирает первые элементы из списков историй. Наконец, тест завершает процедуру, выполняя размещение заказа. Сеансовый компонент с сохранением состояния начинает свой жизненный цикл в момент внедрения в тест и завершает его сразу после окончания фонового процесса, запущенного вызовом метода `placeOrder`. На практике также можно было бы извлечь заказ, размещенный асинхронно, и проверить сохранение результатов в базе данных.

2.3. Использование CDI с компонентами EJB 3

Как отмечалось в главе 1, механизм CDI играет жизненно важную роль в обеспечении надежной поддержки внедрения зависимостей с помощью аннотаций во всех компонентах Java EE, включая EJB 3. В этом разделе мы покажем наиболее типичные способы использования механизма CDI с компонентами EJB 3 – как более надежной замены компонентов, управляемых механизмом JSF, и позволяющего добавлять в EJB компоненты, расположенные на других уровнях, отличных от уровня прикладной логики и не использующие службы EJB непосредственно.